Succinct FSharp

Page containing not just the syntax but walk you through FSharp concepts through examples and ample explanation.

FSharp Interactive (fsi)

Probably the best way to learn F# is using FSharp Interactive (`fsi`). To use it, run `dotnet fsi`.

Things to note:

- To see all commands supported in `fsi`, run `#help;;`
- To quit the tool, run `#quit;;`
- To terminate your input (be it `fsi` command or your F# code), type `;;` at end before pressing Enter key.
- If you press enter after typing `#quit`, then `fsi` would wait for further input even after pressing Enter until you type `;;`.

The `it` value

Run `3 + 2;;` in `fsi`. You'll see the following line:

```
val it: int = 5
```

Results of expressions like `3+2` will be evaluated and stored in a **value** called `it`.

In the above output, `it` has a type `int`.

You can reuse the `it` value:

```
> 3 + 2;;
val it: int = 5

> it + 10;;
val it: int = 15
```

You cannot try add `"10"` to `it` value above as `it`'s type is `int` and `"10"` is of `string` type.

The `+` operator accepts two values of the same type:

```
> it + "10";;

it + "10";;
  -----^^^^

stdin(4,6): error FS0001: The type 'string' does not match the type 'int'
```

- `it` value cannot be mutated:

```
> it <- 23 ;;

  it <- 23 ;;
  ^^^^^^^^

stdin(6,1): error FS0027: This value is not mutable. Consider using the mutable keyword, e.g. 'let mutable it = expression'.
```

Bindings and `let` keyword

Binding is associating a name with a value or a function. You cannot change the value or function associated with the name.

- This would be similar to declaring and initializing a constant variable in languages such as C++ or Java.
- For example, to create a value storing sum of `2` and `3`, you can create the following binding using `let` keyword:

```
> let sum = 2 + 3;;
val sum: int = 2 + 3
```

Try assigning `10` to `sum` as you do in languages like C:

```
> sum = sum + 10;;
val it: bool = false
```

and you will see a surprising output saying boolean value `false` has been saved to `it` value. This is because in F# `=` is only used to compare values.

To overwrite value stored in `sum` value, use `<-` operator. However, this won't work because `let` bindings are **immutable** by default, meaning once value is bind to a name, we cannot change it:

```
> sum <- 10;;

sum <- 10;;
  ^^^^^^^^

stdin(6,1): error FS0027: This value is not mutable. Consider using the mutable keyword, e.g. 'let mutable sum = expression'.
```

Variables using `let mutable`

To create a mutable **variable**, we can use `let mutable`:

```
> let mutable sum = sum;;
val mutable sum: int = 5
```

Now, you can change the value of `sum` using `<-`:

```
> sum <- 10;;
val it: unit = ()

> sum;;
val it: int = 10
```

Notice how we can reuse the `sum` binding defined earlier. F# allows us to reuse a binding/variable name. This also allows us to assign different type of data to `sum`:

```
> sum;;
val it: int = 10


> let sum = "hello";;
val sum: string = "hello"
```

`let..in` vs `let`

`let` is actually a lightweight syntax for `let..in`. For example, the following code with `let`:

```
> let a = 2 ;;
val a: int = 2

> a + 10 ;;
val it: int = 12
```

can be rewritten using `let in` as:

```
> let a = 2 in
- a + 10 ;;
val a: int = 2
val it: int = 12
```

The above code is equivalent to:

- defining a function with parameter `a` and returns value `a + 10`, and later
- invoking this function with argument 2:

```
> (fun a -> a + 10) 2 ;;
val it: int = 12
```

`fun a -> a + 10` above is called a lambda or anonymous function. Functions will be discussed in detail later.

Double-tick identifiers

You can create binding names containing space and symbols like `-` using double-tick identifiers (``):

```
> let ``answer to question of life, universe, and everything`` =
    42;;
val ``answer to question of life, universe, and everything`` : int = 42
```

Whitespace in F#

F# code is whitespace sensitive. It uses whitespace for a block of code unlike languages such as C which use curly braces `{..}`.

You could write your let binding as:

```
let p = "This is a very long string."
```

or in a separate block:

```
let p =
    "This is a very long string."
```

The indentation is done using space. DO NOT use tabs or else F# compiler would throw error. However, if you want to use tabs, then configure IDE to replace tabs with fixed amount of space. F# recommends four spaces for indentation.

Types

Data types in F# are called just **types**. Following are the categories of types in F#:

- Primitive type
- Collections
- Discriminated Unions
- Records
- Classes and Interfaces

Discriminated Unions, Records, Classes and Interfaces will be discussed later.

Primitive Types

Primitive types are the most fundamental types in F#. They are the following:

- `bool` - Possible values are `true` or `false`

  ```
  > let boolVal = true;;
  val boolVal: bool = true
  ```

- `byte` - Values from `0uy` to `255uy`. Notice the suffix `uy` - its required to indicate that the number represents byte value.

  ```
  > let byteVal: byte = 12uy;;
  val byteVal: byte = 12uy
  ```

- `int` - Type to store Integer number. Accepts values from -2,147,483,648 to 2,147,483,647.

  ```
  > let intVal = 12;;
  val intVal: int = 12
  ```

- `float`, `double` - A 64-bit floating point type.

  ```
  > let floatVal = 1.0;;
  val floatVal: float = 1.0
  ```

  `double` is a type alias for `float` (meaning `double` is same as `float` but with different name) and can also be used to declare 64-bit floating point data type:

  ```
  > let doubleVal: double = 1.0;;
  val doubleVal: double = 1.0
  ```

- `float32`, `single` - A 32-bit floating point type.

```
> let float32Val = 1.0f;;
val float32Val: float32 = 1.0f
```

`single` is a type alias for `float32` and can be used to declare 32-bit floating point type:

```
> let singleVal: single = 1.0f;;
val singleVal: single = 1.0f
```

- `char` - Unicode character values.

```
> let charVal = 'c';;
val charVal: char = 'c'
```

- `string` - String Unicode text.

```
> let helloStr = "hello";;
val helloStr: string = "hello"
```

- `unit` - Indicates the absence of an actual value. The type has only one formal value, which is denoted `()`. The unit value, `()`, is often used as a placeholder where a value is needed but no real value is available or makes sense.

```
> let unitVal = ();;
val unitVal: unit = ()
```

There are others as well. For full list, look up offical docs.

String type

- To concatenate strings:

```
/// Create a string using string concatenation
let hello = "Hello" + " World"
```

- Adding `@` in front of a string makes it a **verbatim** string, meaining any escape sequences are ignored, except two quotation mark characters `""` are considered as one quotation mark character `"`:

```
> let p = @"hello\t""world""" ;;
val p: string = "hello\t"world""
```

- Write multi-line strings using triple quotes `"""`:

```
> let multiLineString = """
    <book title="Paradise Lost">
""";;
val multiLineString: string = "
    <book title="Paradise Lost">
"
```

Observe that the string stored in `multiLineString` preserve leading whitespace and requires no escaping for double quotes `"`.

- To create multi-line strings without leading whitespace, use backslash:

```
> let multiLineStr2 =
    "This is a multi-line string \
    without leading \
    whitespace.";;
val multiLineStr2: string =
  "This is a multi-line string without leading whitespace."

> multiLineStr2;;
val it: string = "This is a multi-line string without leading whitespace."
```

Interpolated strings

Interpolated strings are strings that allow you to embed F# expressions into them. For example, you can create a string with F# bindings as follows:

```
> let name = "Sumeet";;
val name: string = "Sumeet"

> $"My name is {name}";;
val it: string = "My name is Sumeet"
```

Interpolated strings start with `$` followed by the string within double quotes (`$"..."`) or triple quotes (`$"""..."""`).

You can format values in an interpolated string using `%(format-specifier)`:

```
> $"%0.4f{System.Math.PI}";;
val it: string = "3.1416"
```

Tuples

A tuple is a grouping of unnamed but ordered values, possibly of different types.

For example:

```
> let tupleVal = (12, 4.5, "Hello");;
val tupleVal: int * float * string = (12, 4.5, "Hello")
```

- Tuple's data type is represented as `type1 * type2 * ...`. For example, tuple `tupleVal` above has the type `int * float * string`.
- Tuple can hold multiple values of different data types. For instance, all values are different in `tupleVal`.
- In place of just values, tuples can also hold results of expressions:

```
> let name = "Hello ";;
val name: string = "Hello "

> let tuple2 = (name + "World", 12 + 30);;
val tuple2: string * int = ("Hello World", 42)
```

- You can **deconstruct** a tuple into separate values and store them in different bindings as follows:

```
> let (helloWorld, answer) = tuple2;;
val helloWorld: string = "Hello World"
val answer: int = 42
```

In case, you wanted to get just first value of `tuple2` and ignore the second, you could use `_` wildcard in place of those values you want to ignore:

```
> let (helloWorld, _) = tuple2;;
val helloWorld: string = "Hello World"
```

The first and second elements of a tuple can be obtained using `fst`, `snd` functions:

```
> tuple2 ;;
val it: string * int = ("Hello World", 42)

> fst tuple2 ;;
val it: string = "Hello World"

> snd tuple2 ;;
val it: int = 42
```

For third or further elements, use pattern matching (discussed later). Or, you could write a function like this:

```
> let third (_, _, c) = c ;;
val third: 'a * 'b * c: 'c -> 'c

> let tuple3 = (fst tuple2, snd tuple2, 5.6) ;;
val tuple3: string * int * float = ("Hello World", 42, 5.6)

> third tuple3 ;;
val it: float = 5.6
```

Quick note about `third`'s function signature - F# automatically converts types of parameters into a generic one like `'a`, `'b` if the type cannot be inferred.

Collections

Collections are data structures which can store multiple values of one or more data types.

Lists

A *list* is an immutable collection of elements of the same type.

- To write a list in a single line:

```
> let list1 = ["a"; "b"]
val list1: string list = ["a"; "b"]
```

- Write a list in multiple lines:

```
> let list2 =
    [
        "a"
        "b"
    ];;
val list2: string list = ["a"; "b"]
```

- Prepending an item to a list using cons (`::`) operator:

```
> let list2 = "c" :: list1;;
val list3: string list = ["c"; "a"; "b"]
```

- Concatenate two lists using `@` operator:

```
> let list4 = list1 @ list2;;
val list4: string list = ["a"; "b"; "a"; "b"]
```

- Generate a range of numbers in ascending order using range (`..`) operator:

```
> [1..5] ;;
val it: int list = [1; 2; 3; 4; 5]
```

- Ranges can have custom interval:

```
> [0..3..30] ;;
val it: int list = [0; 3; 6; 9; 12; 15; 18; 21; 24; 27; 30]

> [0..-3..-30] ;;
val it: int list = [0; -3; -6; -9; -12; -15; -18; -21; -24; -27; -30]
```

As observed above, ranges are end-inclusive and start-inclusive.

Arrays

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements.

- Write an array of elements:

```
> let array1 = [|"a"; "b"|];;
val array1: string[] = [|"a"; "b"|]
```

- Access elements using index number:

```
> array1[0];;
val it: string = "a"
```

- Mutate array elements using `<-` operator:

```
> array1[0] <- "mutated";;
val it: unit = ()

> array1;;
val it: string[] = [|"mutated"; "b"|]
```

Sequences

A *sequence* is a logical series of elements of the same type. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used.

```
> let seq1 =
    seq {
        yield 1
        yield 2

        yield! [5..10]
    }
;;
val seq1: seq<int>
```

- `yield` adds one element to the sequence.
- `yield!` adds a whole subsequence. For instance, `yield! [5..10]` adds element 5 till 9 to the sequence.

Slices

A slice is a subset of any data type. Slices use range (`..`) operator.

For example, suppose you have a list called `tenNums` containing numbers from 1 till 10:

```
> let tenNums = [1 .. 10] ;;
val tenNums: int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

You can get various subsets of `tenNums` using slices:

- Slice of elements for indices `1..5` (5 inclusive):

```
> let slice = tenNums[1 .. 5] ;;
val slice: int list = [2; 3; 4; 5; 6]
```

Note that in F#, index starts from 0.

- Slice of elements from beginning to index 5 (5 inclusive):

```
> let slice = tenNums[ .. 5] ;;
val slice: int list = [1; 2; 3; 4; 5; 6]
```

- Slice of elements from index 5 till end of the list:

```
> let slice = tenNums[5 .. ] ;;
val slice: int list = [6; 7; 8; 9; 10]
```

Slicing works on arrays too:

```
> let tenNumsArray = [| 1 .. 10 |];
val tenNumsArray: int[] = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]

> let arraySlice = tenNumsArray[1..5] ;;
val arraySlice: int[] = [|2; 3; 4; 5; 6|]
```

You can also slice 2-D arrays:

```
> let A = array2D [[1;2;3];[4;5;6];[7;8;9]] ;;
val A: int[,] = [[1; 2; 3]
                 [4; 5; 6]
                 [7; 8; 9]]

> // Take the first row
- let row0 = A[0,*] ;;
val row0: int[] = [|1; 2; 3|]

> // Take all rows but first two column
- let custom = A[*,0..1] ;;
val custom: int[,] = [[1; 2]
                      [4; 5]
                      [7; 8]]
```

Comments

Block comments are placed between `(*` and `*)`.

```
(*
    This is a
    block comment
*)
```

Line comments start from `//` and continue until the end of the line.

```
// This is a line comment
```

XML doc comments come after `///` allowing us to use XML tags to generate documentation.

```
/// The `let` keyword defines an (immutable) value
let result = 1 + 1 = 2
```

Conditional code using `if..then..else`

In F#, `if..then..else` are expressions, meaning it evaluates to some value which can be assigned to a binding.

For example, if you want to compare two numbers, you can write a function that does this using `if..then..else:

```
> let compare x y =
    if x < y then $"{x} is less than {y}"
    else if x > y then $"{x} is greater than {y}"
    else $"{x} and {y} are equal"
;;
val compare: x: 'a -> y: 'a -> string when 'a: comparison

> compare 3 6 ;;
val it: string = "3 is less than 6"

> compare 300 6 ;;
val it: string = "300 is greater than 6"

> compare 6 6 ;;
val it: string = "6 and 6 are equal"
```

`for..in` expression

`for..in` can be used to loop over values in a list:

```
> let numbers = [1; 2; 3] ;;
val numbers: int list = [1; 2; 3]

> for num in numbers do
    printf $"{num}; "
printfn ""
;;
1; 2; 3;
val it: unit = ()
```

You can use it to loop over a range of characters:

```
> for c in 'a' .. 'f' do
    printf $"char {c} - "
printfn ""
;;
char a - char b - char c - char d - char e - char f -
val it: unit = ()
```

You can also loop over a range of numbers with custom interval:

```
> for num in 3..3..24 do
    printf $"{num} - "
printfn ""
;;
3 - 6 - 9 - 12 - 15 - 18 - 21 - 24 -
val it: unit = ()
```

You can also generate a list of numbers:

```
> let multiplesOf3 = [ for num in 3..3..24 do num ] ;;
val multiplesOf3: int list = [3; 6; 9; 12; 15; 18; 21; 24]
```

Beginning and end of a range can also be functions:

```
> let add x y = x + y ;;
val add: x: int -> y: int -> int

> for num in (add 1 2) .. (add 0 3) .. (add 15 3) do
    printf $"{num} - "
printfn ""
;;
3 - 6 - 9 - 12 - 15 - 18 -
val it: unit = ()
```

You can also ignore loop element using wildcard `_`:

```
> let nums = [1; 2; 3] ;;
val nums: int list = [1; 2; 3]

> for _ in nums do printf "Hello! "
printfn ""
;;
Hello! Hello! Hello!
val it: unit = ()
```

`for..to` expression

`for..to` can be used to iterate over a range of value by incrementing counter by 1.

For example, you can loop over values from 1 till 10 as:

```
> for i = 1 to 10 do
    printf $"{i} "
printfn ""
;;
1 2 3 4 5 6 7 8 9 10
val it: unit = ()
```

Notice that the start `1` and end `10` are included in the range.

You could also decrement counter value by 1 using `downto` instead of `to`:

```
> for i = 10 downto 1 do
    printf $"{i} "
printfn ""
;;
10 9 8 7 6 5 4 3 2 1
val it: unit = ()
```

One cool application of this expression is that you can generate a list containing a range of numbers:

```
> let p = [ for i = 10 to 20 do i ] ;;
```

```
val p: int list = [10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20]
```

`while..do` expression

Similar to `for..in` but gives you greater flexibility in incrementing/decrementing value of counter.

For example, you can print even numbers from 10 till 20 using `while..do` as:

```
> let mutable num = 10;;
val mutable num: int = 10

> while num <= 20 do
    printf $"{num} "
    num <- num + 2
printfn ""
;;
10 12 14 16 18 20
val it: unit = ()
```

You can also use `while..do` to generate lists. For instance, you can generate a list containing even numbers from 10 till 20 as:

```
> let mutable num = 10;;

> let p = [ while num <= 20 do num <- num + 2; num - 2 ] ;;
val p: int list = [10; 12; 14; 16; 18; 20]
```

Functions

The `let` keyword also defines named functions.

For example, we will define a function named `negate` which would accept a number and multiply it with `-1` and return the result:

```
> let negate x = x * -1 ;;
val negate: x: int -> int
```

Notice how we didn't provide any data type to the input parameter `x` yet F# figured out `int` data type from the function definition. This is yet another example of powerful **type inference** in F#.

Sometimes, F# cannot correctly infer types, or sometimes you just want to add types for clarity. In those cases, you can provide the types as:

```
> let negate (x: int): int = x * -1 ;;
val negate: x: int -> int
```

Here, `(x: int)` declares the data type of `x`, while as `: int` at the end declares the return type of `negate` function.

Lambda (or Anonymous) functions

You can create anonymous/Lambda functions as follows:

```
> let negateLambda = fun x -> x * -1 ;;
val negateLambda: x: int -> int

> negateLambda 4 ;;
val it: int = -4
```

In F#, lamdbas capture the values of bindings/variables in the scope in which it is defined. Meaning, if `valX` is defined before defining lambda function, then we can use `valX` inside the lambda function definition:

```
> let valX = 3;;
val valX: int = 3

> let negateLambda = fun x -> x * -1 * valX ;;
val negateLambda: x: int -> int

> negateLambda 4 ;;
val it: int = -12
```

Curried Functions

Let's write a function to add two numbers:

```
> let add x y: int = x + y ;;
val add: x: int -> y: int -> int
```

Notice the signature of `add` function: `add: x: int -> y: int -> int`. This can be made sense of by understanding *currying*.

Currying is a process that transforms a function that has more than one parameter into a series of embedded functions, each of which has a single parameter.

So, in the context of `add` function, how F# looks at it is as follows:

```
> let add =
    fun x ->
        fun y ->
            x + y ;;
val add: x: int -> y: int -> int
```

What this means is that F# converted `add` function into a function of one paramter (`fun x ->`) which returns a function of one paramter (`fun y ->`).

- Since type of `x` and `y` is `int`, you can now understand why `add` function signature is `x: int -> y: int -> int`, where `y: int -> int` indicates a function which accepts a parameter `y` of type `int` and returns value of type `int`.
- Similarly, `x: int -> y: int -> int` indicates a function which accepts a parameter `x` of type `int` and returns a function of type `y: int -> int`.

So now, you can create a function called `add2` which would always add `2` to an integer argument:

```
> let add2 = add 2 ;;
val add2: (int -> int)

> add2 10 ;;
val it: int = 12
```

Pipe and composition operators

Pipe operator `|>` is used to chain functions and arguments together:

```
> let square x = x * x ;;
val square: x: int -> int

> let negate x = x * -1 ;;
val negate: x: int -> int

> let ``square and negate`` x =
    x |> square |> negate ;;
val ``square and negate`` : x: int -> int
```

`x |> square |> negate` is equivalent to `negate (square x)`:

```
> ``square and negate`` 10 ;;
val it: int = -100

> negate (square 10) ;;
val it: int = -100
```

You could also write `` ``square and negate`` `` function using composition operator `>>`:

```
> let squareNegate = square >> negate ;;
squareNegate 10;;
```

`>>` operator is used to compose functions - `square >> negate` is equivalent to feeding `square` function's result to `negate` function, or `negate (square x)`.

Recursive functions

The `rec` keyword is used together with the `let` keyword to define a recursive function:

```
let rec factorial x =
    if x < 1 then 1
    else x * factorial (x - 1)
```

Without `rec`, the compiler won't be able to find `factorial` function and hence will throw an error.

Mutually recursive functions (those functions which call each other) are indicated by `and` keyword:

```
let rec even x =
    if x = 0 then true
    else odd (x - 1)

and odd x =
    if x = 0 then false
    else even (x - 1)
```

Pattern Matching

Pattern matching is often facilitated through `match` keyword.

```
// a recursive function to calculate nth fibonacci number
let rec fibonacci n =
    match n with
    // match 0 pattern
    | 0 -> 0
    | 1 -> 1
    // wildcard pattern _
    | _ -> fib (n - 1) + fib (n - 2)
```

- `| 0 -> 0` means when `n` equals `0`, then return `0`
  - `| 0` is called matching pattern `0` or matching arm `0`
  - Pattern and result expression are seperated by `->`
- In pattern matching, all patterns should be matched against, or else F# compiler would give a warning. To add a default pattern to match against any pattern not matched by existing patterns, use wildcard pattern `_`.
  - In the above recursive function, `| _ ->` matches any value of `n` which is not 1 or 0.

`when` keyword

In order to match sophisticated inputs, one can use `when` to create filters or guards on patterns:

```
let sign x =
    match x with
    | 0 -> 0
    // this arm will be matched when x is less than 0
    | x when x < 0 -> -1
    | _ -> 1
```

`function` keyword

`function` keyword is used to simply writing lambda functions containing `match` expression as function body. For instance, you have this lambda expression to compute distance from origin in 1-D, 2-D and 3-D coordinate system:

```
> let distanceFromOrigin =
    fun coordinate ->
        match coordinate with
        | [ x ] -> x
        | [ x; y ] -> System.Math.Sqrt ( x * x + y * y )
        | [ x; y; z ] -> System.Math.Sqrt ( x * x + y * y + z * z )
        | _ -> printfn "Not supported!"; -1
;;
val distanceFromOrigin: coordinate: float list -> float

> distanceFromOrigin [ 3; 4 ] ;;
val it: float = 5.0
```

You could simplify `distanceFromOrigin` function definition using `function` keyword as follows:

```
> let distanceFromOrigin =
    function
        | [ x ] -> x
        | [ x; y ] -> System.Math.Sqrt ( x * x + y * y )
        | [ x; y; z ] -> System.Math.Sqrt ( x * x + y * y + z * z )
```

```
        | _ -> printfn "Not supported!"; -1
;;

val distanceFromOrigin: _arg1: float list -> float

> distanceFromOrigin [ 3; 4 ] ;;
val it: float = 5.0
```

So, `function` keyword basically replaces `fun x -> match x with`.

Cons (`::`) pattern with lists

We can write a recursive function `listSum` which computes sum of numbers in a list as:

```
> let rec sum list =
    match list with
    | [] -> 0
    | head :: tail -> head + sum tail
;;
val sum: list: int list -> int

> sum [] ;;
val it: int = 0

> sum [1; 2; 3; 4] ;;
val it: int = 10
```

- `head :: tail` is called Cons pattern. Using this, we can extract head of the list `head` and rest of the list `tail`.

Active Patterns

Active Pattern helps you to categorize an input data into some named partition, so that you can use these names as a pattern in match expressions.

For example, you can write an active pattern which would classify a number as even or odd as follows:

```
> let (|Even|Odd|) num = if num % 2 = 0 then Even else Odd ;;
val (|Even|Odd|) : num: int -> Choice<unit,unit>
```

- The `(|` and `|)`symbols are called **banana clips**.
- The function created above is called an **active recognizer**. It takes an input an returns an object of type `Choice`.

Now, you can match against a number as even or odd as follows:

```
> let testNum num =
    match num with
    | Odd -> printfn "Number is odd"
    | Even -> printfn "Number is even"
;;
val testNum: num: int -> unit

> testNum 6 ;;
Number is even
val it: unit = ()

> testNum 77 ;;
Number is odd
val it: unit = ()
```

Parameterized active patterns

The above active patterns took just the input. You can provide more arguments to active patterns using **parameterized active patterns**.

For example, you can write an active pattern which would categorize an input `num` based on whether it is divisible by another number `by` as follows:

```
> let (|DivisibleBy|_|) by num =
    if num % by = 0 then Some DivisibleBy else None
;;
val (|DivisibleBy|_|) : by: int -> num: int -> unit option
```

You can use this active pattern in `match` expressions to implement FizzBuzz solution where we check what to do based on whether input is divisible by 3 or 5:

```
> let fizzbuzz =
    function
        | DivisibleBy 5 & DivisibleBy 3 -> printfn "FizzBuzz"
        | DivisibleBy 5 -> printfn "Buzz"
        | DivisibleBy 3 -> printfn "Fizz"
        | x -> printfn "%d" x
;;
val fizzbuzz: _arg1: int -> unit

> fizzbuzz 15;;
FizzBuzz
val it: unit = ()

> fizzbuzz 11;;
11
val it: unit = ()
```

Partial Active Patterns

Sometimes, you need to partition only part of the input space. In that case, you write a set of partial patterns each of which match some inputs but fail to match other inputs. Active patterns that do not always produce a value are called **partial active patterns**; they have a return value that is an `option` type.

To define a partial active pattern, you use a wildcard character (_) at the end of the list of patterns inside the banana clips.

For example, suppose you want to parse a number from a string input, but would like to round float number to nearest integer.

Records

Records represent simple aggregates of named values, optionally with members.

To declare a record type:

```
> type Person = { Name : string; Age : int } ;;
type Person =
  {
    Name: string
    Age: int
  }
```

To create a record via record expression:

```
> let paul = { Name = "Paul"; Age = 28 } ;;
val paul: Person = { Name = "Paul"
                     Age = 28 }
```

Expression to copy and update a record:

```
> let paulsTwin = { paul with Name = "Jim" } ;;
val paulsTwin: Person = { Name = "Jim"
                          Age = 28 }
```

Records can be augmented with properties and methods:

```
> type Person with
    member x.Info = (x.Name, x.Age)
;;
type Person with
  member Info: string * int

> paul.Info ;;
val it: string * int = ("Paul", 28)

> paulsTwin.Info ;;
val it: string * int = ("Jim", 28)
```

Properties of records, such as `Name` in `Person` record type, can't be modified by default. If you want a mutable record property, add `mutable` keyword to it:

```
> type Employee =
    {
        Name: string
        mutable DepartmentId: int
    }
;;
type Employee =
  {
    Name: string
    mutable DepartmentId: int
  }

> let mutable empShyam = { Name = "Shyam"; DepartmentId = 15 } ;;
val mutable empShyam: Employee = { Name = "Shyam"
                                   DepartmentId = 15 }

> // change the department ID for Shyam
- empShyam.DepartmentId <- 20 ;;

> empShyam ;;
val it: Employee = { Name = "Shyam"
                     DepartmentId = 20 }
```

- Note that in order to mutate a record, you also need to declare the record as mutable (e.g. `let mutable empShyam` in above code)

Records are essentially sealed classes with extra topping: default immutability, structural equality, and pattern matching support.

```
> let isPaul person =
    match person with
    | { Name = "Paul"; Age = _ } -> true
    | _ -> false
;;
val isPaul: person: Person -> bool

> isPaul paul ;;
val it: bool = true

> isPaul paulsTwin ;;
val it: bool = false
```

Anonymous Records

Anonymous records are like records but don't need to be declared upfront. For example, here's a function returning an anonymouus record containing diameter, area, circumference of a circle for a given radius:

```
> let getCircleProps radius =
    let dia = radius * 2.0
    // To compute square of radius, you can use ** operator
    let area = System.Math.PI * (radius ** 2.0)
    let peri = 2.0 * System.Math.PI * radius

    {| Diameter = dia; Area = area; Circumference = peri |}
;;
val getCircleProps:
  radius: float -> {| Area: float; Circumference: float; Diameter: float |}
```

You can accept an anonymous record as a parameter:

```
> let circleProps = getCircleProps 4.0 ;;
val circleProps: {| Area: float; Circumference: float; Diameter: float |} =
  { Area = 50.26548246
    Circumference = 25.13274123
    Diameter = 8.0 }

> let printCircleProps (circleProps: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn $"""
        Area: {circleProps.Area}
        Circumference: {circleProps.Circumference}
        Diameter: {circleProps.Diameter}
    """
;;

> printCircleProps circleProps ;;

        Area: 50.26548245743669
        Circumference: 25.132741228718345
        Diameter: 8

val it: unit = ()
```

You can also copy and extend an existing anonymous record:

```
> let circlePropsWithName = {| circleProps with Name = "Circle A" |} ;;
val circlePropsWithName:
```

```
{| Area: float; Circumference: float; Diameter: float; Name: string |} =
  { Area = 50.26548246
    Circumference = 25.13274123
    Diameter = 8.0
    Name = "Circle A" }
```

Discriminated Unions (DU)

Discriminated Unions are helpful to model heterogenous data, or data which can be grouped under same category but can be modeled as different types like `int`, a class or a record.

For example, if you want to model a geometric shape which could be either a square, rectangle or a circle, you could do so using DU:

```
> type Shape =
    | Circle of radius: int
    | Square of side: int
    | Rectange of length: int * width: int
;;
type Shape =
  | Circle of radius: int
  | Square of side: int
  | Rectangle of length: int * width: int
```

Now, you create an instance of `Circle`, `Square` or `Rectangle` type and save it to a binding/variable of type `Shape`:

```
> let shape: Shape = Circle 5 ;;
val shape: Shape = Circle 5

> shape <- Rectangle (5, 3);;
val it: unit = ()

> shape;;
val it: Shape = Rectange (5, 3)
```

`option` type

`option` type is a discriminated union available in F# by default. Options can store a value of some type, or it does not have that value.

`option` is defined as:

```
type 'a option =
    | Some of 'a
    | None
```

`'a` is a generic type parameter which could be any type. `option` type defines two types:

- `Some` type which stores a value of type `'a`
- `None` type which indicates that the `option` does not contain any value

For example, suppose you write a function which accepts a number as an argument and returns that number if it is a positive number or no value otherwise. Such a function could be written using `option` as return type:

```
> let getPositiveNumber num =
    if num > 0 then Some num
    else None
;;
val getPositiveNumber: num: int -> int option

> getPositiveNumber 12;;
val it: int option = Some 12

> getPositiveNumber -12;;
val it: int option = None
```

- Note that the function signature of `getPositiveNumber` is `int -> int option`, meaning the function accepts an `int` value and returns an `option` of type `int`.

How do we extract a value from an `option` if the value exists? For example, suppose we write a function which accepts salary option as a parameter, and we return the salary value if it exists or default salary of `1000` if it does not. We can use pattern matching for this:

```
> let getSalary (salary: int option) =
    match salary with
    | Some salaryVal -> salaryVal
    | None -> 1000
;;
val getSalary: salary: int option -> int

> getSalary (Some 2000) ;;
val it: int = 2000

> getSalary (None) ;;
val it: int = 1000
```

We could also use `Option.defaultValue` function from `Option` module:

```
> let getSalary (salary: int option) =
    salary |> Option.defaultValue 1000
;;

> getSalary (None) ;;
val it: int = 1000
```

There are lots of functions available in `Option` module which deal with `option` types. For full list, visit this page

`Result` type

`Result` type can be used to handle errors in a typesafe way. A binding of `Result` type can accept either `Ok` containing success data of any type, or `Error` containing error data of any type.

For example, you can write a function which divides two numbers using `Result` type as follows:

```
> let divide x y =
    if y = 0 then Error "Trying to divide by 0"
    else Ok (x / y)
;;
val divide: x: int -> y: int -> Result<int,string>

> divide 6 3 ;;
val it: Result<int,string> = Ok 2

> divide 6 0 ;;
val it: Result<int,string> = Error "Trying to divide by 0"
```

You can pattern match against the result value to perform action based on whether the operation was successful or failed due to an error:

```
> let res = divide 6 3 ;;

> match res with
    | Ok result -> printfn $"The result of 6/3 is {result}"
    | Error message -> printfn $"An error occured"
;;
The result of 6/3 is 2
val it: unit = ()
```

Generics and Automatic Generalization

Consider the following function definition:

```
> let makeList a b =
    [a; b]
;;
val makeList: a: 'a -> b: 'a -> 'a list
```

You see that F# infers `makeList` signature as `'a -> 'a -> 'a list`. Two takeaways from this:

- F# automatically generalized the function in the event no type could be inferred from the definition. This is called as **Automatic Generalization**.
- Since a list could have elements of the same type, both `a` and `b` have been assigned the same generic type `'a`.
- Generic types follow the convention: `'name`. For e.g., `'a`, `'hello`.

You can explicitly specify generics as:

```
> let makeList (a: 'a) (b: 'a) =
    [a; b]
;;
val makeList: a: 'a -> b: 'a -> 'a list
```

You can make it even more explicit using `<'generic>`:

```
> let makeList<'a> (a: 'a) (b: 'a) =
    [a; b]
;;
val makeList: a: 'a -> b: 'a -> 'a list
```

Exceptions

failwith

The `failwith` function throws an exception of type `Exception`.

```
> let divideFailwith x y =
    if y = 0 then
        failwith "Divisor cannot be zero."
    else x / y
;;
val divideFailwith: x: int -> y: int -> int

> divideFailwith 15 0;;
System.Exception: Divisor cannot be zero.
   at FSI_0060.divideFailwith(Int32 x, Int32 y)
   at <StartupCode$FSI_0061>.$FSI_0061.main@()
Stopped due to error
```

try/with

Exception handling is done via `try/with` expressions.

```
> let divide x y =
    try
        Some (x / y)
    with :? System.DivideByZeroException ->
        printfn "Division by zero!"
        None
;;

> divide 6 3;;
val it: int option = Some 2

> divide 5 0;;
Division by zero!
val it: int option = None
```

- Exception thrown is tried to be downcasted to `System.DivideByZeroException` using :?. If it is successful, then the `"Division by zero!"` message is printed and function returns `None`.

Custom exceptions

You can throw custom exceptions by first defining them using `exception` keyword:

```
> exception CustomDivideByZero of string ;;
exception CustomDivideByZero of string
```

The above exception `CustomDivideByZero` accepts only string message as an argument.

You can then throw this custom exception via `raise` function:

```
> let divide x y =
    if y = 0
    then
        raise (CustomDivideByZero ("trying to divide a number by zero!"))
    else
        (x |> float) / (y |> float)
;;
val divide: x: int -> y: int -> float

> divide 6 4;;
val it: float = 1.5
```

```
> divide 6 0;;
FSI_0065+CustomDivideByZero: Exception of type 'FSI_0065+CustomDivideByZero' was thrown.
   at FSI_0071.divide(Int32 x, Int32 y)
   at <StartupCode$FSI_0073>.$FSI_0073.main@()
Stopped due to error
```

`try/finally`

The `try/finally` expression enables you to execute clean-up code even if a block of code throws an exception.

```
> exception InnerError of string ;;
exception InnerError of string

> exception OuterError of string ;;
exception OuterError of string

> let handleErrors x y =
    try
        try
            if x = y
            then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with InnerError(str) ->
            printfn "Error1 %s was caught" str
    finally
        printfn "Always print this."
 ;;
val handleErrors: x: 'a -> y: 'a -> unit when 'a: equality

> handleErrors 3 3 ;;
Error1 inner was caught
Always print this.
val it: unit = ()

> handleErrors 3 4 ;;
Always print this.
FSI_0075+OuterError: Exception of type 'FSI_0075+OuterError' was thrown.
   at FSI_0076.handleErrors[a](a x, a y)
   at <StartupCode$FSI_0078>.$FSI_0078.main@()
Stopped due to error
```

Classes and Inheritance

In F#, you can create classes to model data.

For example, suppose you want to create a data structure to store a vector. You can start with create a class type:

```
type Vector() = class end
```

- `Vector()` provides class name and constructor to call in order to create a `Vector` object via `new Vector()`:

```
let vector = Vector()
```

- `class end` is required if your class doesn't require any properties or methods.

Next, we will add two class properties `X` and `Y` and will define a constructor for it:

```
type Vector (x: float, y: float) =
    member this.X = x
    member _.Y = y
```

- Properties and methods are defined using `member` keyword.
- `this` can be replaced with wildcard `_` if it is not required. Also, you could use `self` or any other name in place of `this` because unlike Java, `this` is not a keyword in F#.

You can add getters and setters for `Y` property as follows:

```
let mutable yVal = y
member _.Y
    with get() = yVal
    and set(newVal) = yVal <- newVal
```

- Important - `let mutable yVal` binding must come before any `member` declarations. So, if the first `member` declaration is `member this.X`, then `yVal` binding definition should come before it.

We can add methods to `Vector` clas definition like `Scale` to scale the vector:

```
member this.Scale scaleVal =
    Vector(this.X * scaleVal, this.Y * scaleVal)
```

- In F#, properties and methods should start with a capital letter.

We can also implement operations such as adding two vectors by implementing `+` operator via static methods:

```
static member (+) (a : Vector, b : Vector) =
    Vector(a.X + b.X, a.Y + b.Y)
```

You can also have `let` bindings inside class definition. Such bindings won't be available outside the class and can be used to compute members' values. For example, we can define a member `mag` which will store the computed magnitude of the vector:

```
type Vector (x: float, y: float) =
    let magnitude = sqrt(x * x, y * y)
    member _.mag = magnitude
```

Notice how value of `this.mag` depends on the constructor arguments `x` and `y`. Since we have a setter for `y`, let's modify `this.mag` to update value when `this.Y` changes:

```
let magnitude() = sqrt(x * x, yVal * yVal)
member _.mag
    with get() = magnitude()
```

Now, everytime you fetch `mag` property, `magnitude` function would be executed.

Overall, our Vector class now looks like this:

```
> type Vector(x : float, y : float) =
    let mutable yVal = y
    let magnitude() = sqrt(x * x + yVal * yVal)
```

```
    member this.X = x
    member _.Y
        with get() = yVal
        and set(newVal) = yVal <- newVal
    member _.mag
        with get() = magnitude()
    member this.Scale scaleVal =
        Vector(this.X * scaleVal, this.Y * scaleVal)
    static member (+) (a : Vector, b : Vector) =
        Vector(a.X + b.X, a.Y + b.Y)
;;
type Vector =
  new: x: float * y: float -> Vector
  member Scale: s: float -> Vector
  static member (+) : a: Vector * b: Vector -> Vector
  member Mag: float
  member X: float
  member Y: float
```

Now, you can use `Vector` class as follows:

```
> // Creating an instance of Vector class
- let v1 = Vector (3.0, 3.0);;
val v1: Vector

> v1.mag;;
val it: float = 4.242640687

> // Setting value of Y property
- v1.Y <- 4.0;;
val it: unit = ()

> v1.Y;;
val it: float = 4.0

> v1.mag;;
val it: float = 5.0

> v1.Scale 10;;
val it: Vector = FSI_0048+Vector {X = 30.0;
                                  Y = 40.0;
                                  mag = 50.0;}

> let v2 = Vector (12.0, 9.0);;
val v2: Vector

> v1 + v2;;
val it: Vector = FSI_0048+Vector {X = 15.0;
                                  Y = 13.0;
                                  mag = 19.84943324;}

> // Trying to access let binding `magnitude`
- v1.magnitude();;
v1.magnitude()
  ---^^^^^^^^^

stdin(304,4): error FS0039: The type 'Vector' does not define the field, constructor or member 'magnitude'.
```

Inheritance

Call a base class from a derived one. For example, we have a base class `Animal` and a derived class `Dog` inheriting from `Animal` class. We can call `Animal` class methods from `Dog` using `base.MethodName(..)` syntax:

```
type Animal() =
    member __.Rest() = ()

type Dog() =
    inherit Animal()
    member __.Run() =
        base.Rest()
```

Upcasting is the process of converting a child class to a parent class. It can be done using `:>` operator.

```
let dog = Dog()
let animal = dog :> Animal
```

Dynamic downcasting is convering a parent class to a child class. It can be done using (`:?>`) operator. This might throw an `InvalidCastException` if the cast doesn't succeed at runtime:

```
let shouldBeADog = animal :?> Dog
```

Interfaces and Object Expressions

Declare `IVector` interface and implement it in `Vector'`.

```
type IVector =
    abstract Scale : float -> IVector

type Vector'(x, y) =
    interface IVector with
        member _.Scale(s) =
            Vector'(x * s, y * s) :> IVector
    member _.X = x
    member _.Y = y
```

Another way of implementing interfaces is to use *object expressions*.

```
type ICustomer =
    abstract Name : string
    abstract Age : int

let createCustomer name age =
    { new ICustomer with
        member __.Name = name
        member __.Age = age }
```

Compiler Directives

Load another F# source file into FSI.

```
#load "../lib/StringParsing.fs"
```

Reference a .NET assembly (`/` symbol is recommended for Mono compatibility).

```
#r "../lib/FSharp.Markdown.dll"
```

Include a directory in assembly search paths.

```
#I "../lib"
#r "FSharp.Markdown.dll"
```

Other important directives are conditional execution in FSI (`INTERACTIVE`) and querying current directory (`__SOURCE_DIRECTORY__`).

```
#if INTERACTIVE
let path = __SOURCE_DIRECTORY__ + "../lib"
#else
let path = "../../../lib"
#endif
```

F# Cheatsheet 

An updated cheatsheet for F#.

This cheatsheet glances over some of the common syntax of F#.

Contents

Comments

Line comments start from `//` and continue until the end of the line. Block comments are placed between `(*` and `*)`.

```
// And this is line comment
(* This is block comment *)
```

XML doc comments come after `///` allowing us to use XML tags to generate documentation.

```

```

```
/// The `let` keyword defines an (immutable) value
let result = 1 + 1 = 2
```

Strings

The F# `string` type is an alias for `System.String` type. See Strings.

```
/// Create a string using string concatenation
let hello = "Hello" + " World"
```

Use *verbatim strings* preceded by `@` symbol to avoid escaping control characters (except escaping `"` by `""`).

```
let verbatimXml = @"<book title=""Paradise Lost"">"
```

We don't even have to escape `"` with *triple-quoted strings*.

```
let tripleXml = """<book title="Paradise Lost">"""
```

Backslash strings indent string contents by stripping leading spaces.

```
let poem =
    "The lesser world was daubed\n\
     By a colorist of modest skill\n\
     A master limned you in the finest inks\n\
     And with a fresh-cut quill."
```

Interpolated strings let you write code in "holes" inside of a string literal:

```
let name = "Phillip"
let age = 30
printfn $"Name: {name}, Age: {age}"

let str = $"A pair of braces: {{}}"
printfn $"Name: %s{name}, Age: %d{age}" // typed
```

Types and Literals

Most numeric types have associated suffixes, e.g., `uy` for unsigned 8-bit integers and `L` for signed 64-bit integer.

```
let b, i, l, ul = 86uy, 86, 86L, 86UL

// val ul: uint64 = 86UL
// val l: int64 = 86L
// val i: int = 86
// val b: byte = 86uy
```

Other common examples are `F` or `f` for 32-bit floating-point numbers, `M` or `m` for decimals, and `I` for big integers.

```
let s, f, d, bi = 4.14F, 4.14, 0.7833M, 9999I

// val bi: System.Numerics.BigInteger = 9999
// val d: decimal = 0.7833M
// val f: float = 4.14
// val s: float32 = 4.14f
```

See Literals for complete reference.

`and` keyword is used for definining mutually recursive types and functions:

```
type A =
  | Aaa of int
  | Aaaa of C
and C =
  { Bbb : B }
and B() =
  member x.Bbb = Aaa 10
```

Floating point and signed integer values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on:

```
[<Measure>] type kg
let m1 = 10.0<kg>
let m2 = m1 * 2.0 // type inference for result
let add30kg m =   // type inference for input and output
    m + 30.0<kg>
add30 2.0<kg>     // val it: float<kg> = 32.0
```

Printing Things

Print things to console with `printfn`:

```
printfn "Hello, World"

printfn $"The time is {System.DateTime.Now}"
```

You can also use `Console.WriteLine`:

```
open System

Console.WriteLine $"The time is {System.DateTime.Now}"
```

Constrain types with `%d`, `%s`, and print structured values with `%A`:

```
let data = [1..10]

printfn $"The numbers %d{1} to %d{10} are %A{data}"
```

Omit holes and apply arguments:

```
printfn "The numbers %d to %d are %A" 1 10 data
```

See Plaintext Formatting

Loops

for...in

For loops:

```
let list1 = [1; 5; 100; 450; 788]

for i in list1 do
    printf "%d" i           // 1 5 100 450 788

let seq1 = seq { for i in 1 .. 10 -> (i, i * i) }

for (a, asqr) in seq1 do
    // 1 squared is 1
    // ...
    // 10 squared is 100
    printfn "%d squared is %d" a asqr

for i in 1 .. 10 do
    printf "%d " i          // 1 2 3 4 5 6 7 8 9 10
```

```
// for i in 10 .. -1 .. 1 do
for i = 10 downto 1 do
    printf "%i " i          // 10 9 8 7 6 5 4 3 2 1

for i in 1 .. 2 .. 10 do
    printf "%d " i              // 1 3 5 7 9

for c in 'a' .. 'z' do
    printf "%c " c          // a b c ... z

// Using of a wildcard character (_)
// when the element is not needed in the loop.
let mutable count = 0

for _ in list1 do
    count <- count + 1
```

while...do

While loops:

```
let mutable mutVal = 0
while mutVal < 10 do        // while (not) test-expression do
    mutVal <- mutVal + 1
```

Functions

The `let` keyword also defines named functions.

```
let pi () = 3.14159 // function with no arguments. () is called unit type
pi ()               // it's necessary to use () to call the function

let negate x = x * -1
let square x = x * x
let print x = printfn $"The number is: %d{x}"

let squareNegateThenPrint x =
    print (negate (square x))
```

Double-backtick identifiers are handy to improve readability especially in unit testing:

```
let ``square, negate, then print`` x =
    print (negate (square x))
```

Pipe operator

The pipe operator `|>` is used to chain functions and arguments together:

```
let squareNegateThenPrint x =
    x |> square |> negate |> print
```

This operator is essential in assisting the F# type checker by providing type information before use:

```
let sumOfLengths (xs : string []) =
    xs
    |> Array.map (fun s -> s.Length)
    |> Array.sum
```

Composition operator

The composition operator `>>` is used to compose functions:

```
let squareNegateThenPrint =
    square >> negate >> print
```

Pattern Matching

Pattern matching is primarily through `match` keyword;

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (n - 1) + fib (n - 2)
```

Use `when` to create filters or guards on patterns:

```
let sign x =
    match x with
    | 0 -> 0
    | x when x < 0 -> -1
    | x -> 1
```

Pattern matching can be done directly on arguments:

```
let fst (x, _) = x
```

or implicitly via `function` keyword:

```
/// Similar to `fib`; using `function` for pattern matching
let rec fib2 = function
    | 0 -> 0
    | 1 -> 1
    | n -> fib2 (n - 1) + fib2 (n - 2)
```

See Pattern Matching.

Collections

Lists

Lists are immutable collection of elements of the same type.

```
// Lists use square brackets and `;` delimiter
let list1 = ["a"; "b"]

// :: is prepending
let list2 = "c" :: list1

// @ is concat
let list3 = list1 @ list2

// Recursion on list using (::) operator
let rec sum list =
    match list with
    | [] -> 0
    | x :: xs -> x + sum xs
```

Arrays

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements.

```
// Arrays use square brackets with bar
let array1 = [| "a"; "b" |]

// Indexed access using dot
let first1 = array1.[0]
let first2 = array1[0]    // F# 6
```

Sequences == IEnumerable

Sequences are logical series of elements of the same type. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements

are used.

```
// Sequences can use yield and contain subsequences
seq {
    // "yield" adds one element
    yield 1
    yield 2

    // "yield!" adds a whole subsequence
    yield! [5..10]
}
```

The `yield` can normally be omitted:

```
// Sequences can use yield and contain subsequences
seq {
    1
    2
    yield! [5..10]
}
```

Mutable Dictionaries (from BCL)

Create a dictionary, add two entries, remove an entry, lookup an entry

```
open System.Collections.Generic

let inventory = Dictionary<string, float>()

inventory.Add("Apples", 0.33)
inventory.Add("Oranges", 0.5)

inventory.Remove "Oranges"

// Read the value. If not exists - throw exception.
let bananas1 = inventory.["Apples"]
let bananas2 = inventory["Apples"]   // F# 6
```

Additional F# syntax:

```
// Generic type inference with Dictionary
let inventory = Dictionary<_,_>()   // or let inventory = Dictionary()

inventory.Add("Apples", 0.33)
```

dict == IDictionary in BCL

dict creates immutable dictionaries. You can't add and remove items to it.

```
open System.Collections.Generic

let inventory : IDictionary<string, float> =
    ["Apples", 0.33; "Oranges", 0.23; "Bananas", 0.45]
    |> dict

let bananas = inventory.["Bananas"]     // 0.45
let bananas2 = inventory["Bananas"]     // 0.45, F# 6

inventory.Add("Pineapples", 0.85)       // System.NotSupportedException
inventory.Remove("Bananas")             // System.NotSupportedException
```

Quickly creating full dictionaries:

```
[ "Apples", 10; "Bananas", 20; "Grapes", 15 ] |> dict |> Dictionary
```

Map

Map is an immutable key/value lookup. Allows safely add or remove items.

```
let inventory =
    Map ["Apples", 0.33; "Oranges", 0.23; "Bananas", 0.45]

let apples = inventory.["Apples"]
let apples = inventory["Apples"] // F# 6
let pineapples = inventory.["Pineapples"]   // KeyNotFoundException
let pineapples = inventory["Pineapples"]    // KeyNotFoundException on F# 6 too

let newInventory =              // Creates new Map
    inventory
    |> Map.add "Pineapples" 0.87
    |> Map.remove "Apples"
```

Safely access a key in a *Map* by using *TryFind*. It returns a wrapped option:

```
let inventory =
    Map ["Apples", 0.33; "Oranges", 0.23; "Bananas", 0.45]

inventory.TryFind "Apples"      // option = Some 0.33
inventory.TryFind "Unknown"     // option = None
```

Useful Map functions include `map`, `filter`, `partition`:

```
let cheapFruit, expensiveFruit =
    inventory
    |> Map.partition(fun fruit cost -> cost < 0.3)
```

Dictionaries, dict, or Map?

- Use *Map* as your default lookup type:
    - It's immutable
    - Has good support for F# tuples and pipelining.
- Use the *dict* function
    - Quickly generate an *IDictionary* to interop with BCL code.
    - To create a full Dictionary.
- Use *Dictionary*:
    - If need a mutable dictionary.
    - Need specific performance requirements. (Example: tight loop performing thousands of additions or removals).

Generating lists

The same list `[ 1; 3; 5; 7; 9 ]` can be generated in various ways.

```
[ 1; 3; 5; 7; 9 ]
[ 1..2..9 ]
[ for i in 0..4 -> 2 * i + 1 ]
List.init 5 (fun i -> 2 * i + 1)
```

The array `[| 1; 3; 5; 7; 9 |]` can be generated similarly:

```
[| 1; 3; 5; 7; 9 |]
[| 1..2..9 |]
[| for i in 0..4 -> 2 * i + 1 |]
Array.init 5 (fun i -> 2 * i + 1)
```

Functions on collections

Lists and arrays have comprehensive functions for manipulation.

- `List.map` transforms every element of the list (or array)
- `List.iter` iterates through a list and produces side effects

These and other functions are covered below. All these operations are also available for sequences.

Tuples and Records

A *tuple* is a grouping of unnamed but ordered values, possibly of different types:

```
// Tuple construction
let x = (1, "Hello")

// Triple
let y = ("one", "two", "three")

// Tuple deconstruction / pattern
let (a', b') = x
```

The first and second elements of a tuple can be obtained using `fst`, `snd`, or pattern matching:

```
let c' = fst (1, 2)
let d' = snd (1, 2)

let print' tuple =
    match tuple with
    | (a, b) -> printfn "Pair %A %A" a b
```

Records represent simple aggregates of named values, optionally with members:

```
// Declare a record type
type Person = { Name : string; Age : int }

// Create a value via record expression
let paul = { Name = "Paul"; Age = 28 }

// 'Copy and update' record expression
let paulsTwin = { paul with Name = "Jim" }
```

Records can be augmented with properties and methods:

```
type Person with
     member x.Info = (x.Name, x.Age)
```

Records are essentially sealed classes with extra topping: default immutability, structural equality, and pattern matching support.

```
let isPaul person =
    match person with
    | { Name = "Paul" } -> true
    | _ -> false
```

Recursive Functions

The `rec` keyword is used together with the `let` keyword to define a recursive function:

```
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)
```

Mutually recursive functions (those functions which call each other) are indicated by `and` keyword:

```
let rec even x =
   if x = 0 then true
   else odd (x - 1)

and odd x =
   if x = 0 then false
   else even (x - 1)
```

Discriminated Unions

Discriminated unions (DU) provide support for values that can be one of a number of named cases, each possibly with different values and types.

```
type Tree<'T> =
    | Node of Tree<'T> * 'T * Tree<'T>
    | Leaf


let rec depth input =
    match input with
    | Node(l, _, r) -> 1 + max (depth l) (depth r)
    | Leaf -> 0
```

F# Core has a few built-in discriminated unions for error handling, e.g., Option and Result.

Using Option:

```
let optionPatternMatch input =
    match input with
```

```
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"

optionPatternMatch (Some 1)
optionPatternMatch None
```

Using Result:

```
let resultPatternMatch input =
    match input with
    | Ok i -> printfn "Success with code %d" i
    | Error e -> printfn "Error with code %d" e

resultPatternMatch (Ok 0)
resultPatternMatch (Error 1)
```

Single-case discriminated unions are often used to create type-safe abstractions with pattern matching support:

```
type OrderId = Order of string

// Create a DU value
let orderId = Order "12"

// Use pattern matching to deconstruct single-case DU
let (Order id) = orderId
```

Exceptions

The `failwith` function throws an exception of type `Exception`.

```
let divideFailwith x y =
    if y = 0 then
        failwith "Divisor cannot be zero."
        else x / y
```

Exception handling is done via `try/with` expressions.

```
let divide x y =
    try
        Some (x / y)
    with :? System.DivideByZeroException ->
        printfn "Division by zero!"
        None
```

The `try/finally` expression enables you to execute clean-up code even if a block of code throws an exception. Here's an example which also defines custom exceptions.

```
exception InnerError of string
exception OuterError of string

let handleErrors x y =
   try
       try
           if x = y then raise (InnerError("inner"))
           else raise (OuterError("outer"))
       with InnerError(str) ->
          printfn "Error1 %s" str
   finally
       printfn "Always print this."
```

Classes and Inheritance

This example is a basic class with (1) local let bindings, (2) properties, (3) methods, and (4) static members.

```
type Vector(x: float, y: float) =
    let mag = sqrt(x * x + y * y)               // (1) - local let binding

    member this.X = x                           // (2) property
    member this.Y = y                           // (2) property
    member this.Mag = mag                       // (2) property

    member this.Scale(s) =                       // (3) method
        Vector(x * s, y * s)

    static member (+) (a : Vector, b : Vector) = // (4) static method
        Vector(a.X + b.X, a.Y + b.Y)
```

Call a base class from a derived one:

```
type Animal() =
    member _.Rest() = ()

type Dog() =
    inherit Animal()
    member _.Run() =
        base.Rest()
```

Interfaces and Object Expressions

Declare `IVector` interface and implement it in `Vector'`.

```
type IVector =
    abstract Scale : float -> IVector

type Vector(x, y) =
    interface IVector with
        member __.Scale(s) =
            Vector(x * s, y * s) :> IVector

    member __.X = x

    member __.Y = y
```

Another way of implementing interfaces is to use *object expressions*.

```
type ICustomer =
    abstract Name : string
    abstract Age : int

let createCustomer name age =
    { new ICustomer with
        member __.Name = name
        member __.Age = age }
```

Casting and Conversions

```
int 3.1415     // float to int = 3
int "3"        // string to int = 3
float 3        // int to float = 3.0
float "3.1415" // string to float = 3.1415
string 3       // int to string = "3"
string 3.1415  // float to string = "3.1415"
```

Upcasting is denoted by `:>` operator.

```
let dog = Dog()
let animal = dog :> Animal
```

In many places type inference applies upcasting automatically:

```
let exerciseAnimal (animal: Animal) = ()

let dog = Dog()
```

```
exerciseAnimal dog   // no need to upcast dog to Animal
```

Dynamic downcasting (`:?>`) might throw an `InvalidCastException` if the cast doesn't succeed at runtime.

```
let shouldBeADog = animal :?> Dog
```

Active Patterns

Complete active patterns:

```
let (|Even|Odd|) i =
  if i % 2 = 0 then Even else Odd

let testNumber i =
    match i with
    | Even -> printfn "%d is even" i
    | Odd -> printfn "%d is odd" i
```

Parameterized, partial active patterns:

```
let (|DivisibleBy|_|) divisor n =
  if n % divisor = 0 then Some DivisibleBy else None

let fizzBuzz input =
    match input with
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz"
    | DivisibleBy 3 -> "Fizz"
    | DivisibleBy 5 -> "Buzz"
    | i -> string i
```

Partial active patterns share the syntax of parameterized patterns but their active recognizers accept only one argument.

Compiler Directives

Load another F# source file into F# Interactive (`dotnet fsi`).

```
#load "../lib/StringParsing.fs"
```

Reference a .NET package:

```
#r "nuget: FSharp.Data"                // latest non-beta version
#r "nuget: FSharp.Data,Version=4.2.2"  // specific version
```

Specifying a package source:

```
#i "nuget: https://my-remote-package-source/index.json"

#i """nuget: C:\path\to\my\local\source"""
```

Reference a specific .NET assembly file:

```
#r "../lib/FSharp.Markdown.dll"
```

Include a directory in assembly search paths:

```
#I "../lib"
#r "FSharp.Markdown.dll"
```

Other important directives are conditional execution in FSI (`INTERACTIVE`), conditional for compiled code (`COMPILED`) and querying current directory (`__SOURCE_DIRECTORY__`).

```
#if INTERACTIVE
let path = __SOURCE_DIRECTORY__ + "../lib"
#else
let path = "../../../lib"
#endif
```

Useful functions

identity function (id)

It's useful for cases where you need a lambda like `fun x -> x`:

```
[[1;2]; [3]] |> List.collect (fun x -> x)  // [1; 2; 3]
[[1;2]; [3]] |> List.collect id            // [1; 2; 3]
```

Mapping functions

map (Array, List, Seq)

Converts all the items in a collection from one shape to another shape. Always returns the same number of items in the output collection as were passed in.

```
// [2; 4; 6; 8; 10; 12; 14; 16; 18; 20]
[1 .. 10] |> List.map (fun n -> n * 2)

type Person = { Name : string; Town : string }

let persons =
    [
        { Name = "Isaak"; Town = "London" }
        { Name = "Sara"; Town = "Birmingham" }
        { Name = "Tim"; Town = "London" }
        { Name = "Michelle"; Town = "Manchester" }
    ]

// ["London"; "Birmingham"; "London"; "Manchester"]
persons |> List.map (fun person -> person.Town)
```

map2, map3 (Array, List, Seq)

map2 and *map3* are variations of *map* that take multiple lists. The collections must have equal lengths, except for `Seq.map2` and `Seq.map3` where extra elements are ignored.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let list3 = [7; 8; 9]

// [5; 7; 9]
(list1, list2) ||> List.map2 (fun x y -> x + y)

// [12; 15; 18]
(list1, list2, list3) |||> List.map3 (fun x y z -> x + y + z)
```

mapi, mapi2 (Array, List, Seq)

mapi and *mapi2* - in addition to the element, the function needs to be passed the index of each element. The only difference `mapi2` and `mapi` is that `mapi2` works with two collections.

```
let list1 = [9; 12; 53; 24; 35]

// [(0, 9); (1, 12); (2, 53); (3, 24); (4, 35)]
list1 |> List.mapi (fun x i -> (x, i))

// [9; 13; 55; 27; 39]
list1 |> List.mapi (fun x i -> x + i)
```

```
let list1 = [9; 12; 3]
let list2 = [24; 5; 2]

// [0; 17; 10]
(list1, list2) ||> List.mapi2 (fun i x y -> (x + y) * i)
```

indexed (Array, List, Seq)

Returns a new collection whose elements are the corresponding elements of the input paired with the index (from 0) of each element.

```
let list1 = [23; 5; 12]

// [(0, 23); (1, 5); (2, 12)]
let result = list1 |> Array.indexed
```

iter, iter2, iteri, iteri2 (Array, List, Seq)

iter is the same as a `for` loop, the function that you pass in must return unit.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]

// Prints: 1; 2; 3;
list1 |> List.iter (fun x -> printf "%d; " x)

// Prints: (1 4); (2 5); (3 6);
(list1, list2) ||> List.iter2 (fun x y -> printf "(%d %d); " x y)

// Prints: ([0] = 1); ([1] = 2); ([2] = 3);
list1 |> List.iteri (fun i x -> printf "([%d] = %d); " i x)

// Prints: ([0] = 1 4); ([1] = 2 5); ([2] = 3 6);
(list1, list2) ||> List.iteri2 (fun i x y -> printf "([%d] = %d %d); " i x y)
```

collect (Array, List, Seq)

collect runs a specified function on each element and then collects the elements generated by the function and combines them into a new collection.

```
// [0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let list1 = [1; 5; 10]
let list2 = [1; 2; 3]

list1 |> List.collect (fun elem -> [0 .. elem])
```

```
// [1; 2; 3; 2; 4; 6; 3; 6; 9]
list2 |> List.collect (fun x -> [for i in 1..3 -> x * i])

// Example 3
type Customer =
    {
        Id: int
        OrderId: int list
    }

let c1 = { Id = 1; OrderId = [1; 2]}
let c2 = { Id = 2; OrderId = [43]}
let c3 = { Id = 5; OrderId = [39; 56]}
let customers = [c1; c2; c3]

// [1; 2; 43; 39; 56]
let orders = customers |> List.collect(fun c -> c.OrderId)
```

pairwise (Array, List, Seq)

pairwise takes a collection and returns a new list of tuple pairs of the original adjacent items.

```
let list1 = [1; 30; 12; 20]

//  [(1, 30); (30, 12); (12, 20)]
list1 |> List.pairwise

// [31.0; 11.0; 21.0]
[ DateTime(2010,5,1)
  DateTime(2010,6,1)
  DateTime(2010,6,12)
  DateTime(2010,7,3) ]
|> List.pairwise
|> List.map(fun (a, b) -> b - a)
|> List.map(fun time -> time.TotalDays)
```

windowed (Array, List, Seq)

Returns a list of sliding windows containing elements drawn from the input collection. Each window is returned as a fresh collection. Unlike *pairwise* the windows are collections, not tuples.

```
// [['a'; 'b'; 'c']; ['b'; 'c'; 'd']; ['c'; 'd'; 'e']]
['a'..'e'] |> List.windowed 3
```

Grouping functions

groupBy (Array, List, Seq) == GroupBy() in LINQ

groupBy works exactly as the LINQ version does. The output is a collection of simple tuples. The first element of the tuple is the key, and the second element is the collection of items in that group.

```
type Person =
    {
        Name: string
        Town: string
    }

let persons =
    [ { Name = "Isaak"; Town = "London" }
      { Name = "Sara"; Town = "Birnmingham" }
      { Name = "Tim"; Town = "London" }
      { Name = "Michelle"; Town = "Manchester" } ]

// [("London", [{ Name = "Isaak"; Town = "London" }; { Name = "Tim"; Town =
"London" }]);
//  ("Birnmingham", [{ Name = "Sara"; Town = "Birnmingham" }]);
//  ("Manchester", [{ Name = "Michelle"; Town = "Manchester" }])]
persons |> List.groupBy (fun person -> person.Town)
```

countBy (Array, List, Seq)

A useful derivative of *groupBy* is *countBy*. This has a similar signature, but instead of returning the items in the group, it returns the number of items in each group.

```
type Person = { Name: string; Town: string }

let persons =
    [
        { Name = "Isaak"; Town = "London" }
        { Name = "Sara"; Town = "Birnmingham" }
        { Name = "Tim"; Town = "London" }
        { Name = "Michelle"; Town = "Manchester" }
    ]

// [("London", 2); ("Birnmingham", 1); ("Manchester", 1)]
persons |> List.countBy (fun person -> person.Town)
```

partition (Array, List)

partition use predicate and a collection; it returns two collections, partitioned based on the predicate:

```
// Tupled result in two lists
let londonPersons, otherPersons =
    persons |> List.partition(fun p -> p.Town = "London")
```

If there are no matches for either half of the split, an empty collection is returned for that half.

chunkBySize (Array, List, Seq)

chunkBySize groups elements into arrays (chunks) of a given size.

```
let list1 = [33; 5; 16]

// int list list = [[33; 5]; [16]]
let chunkedLst = list1 |> List.chunkBySize 2
```

splitAt (Array, List)

splitAt splits an Array (List) into two parts at the index you specify. The first part ends just before the element at the given index; the second part starts with the element at the given index.

```
let xs = [| 1; 2; 3; 4; 5 |]

let left1, right1 = xs |> Array.splitAt 0   // [|||] and [|1; 2; 3; 4; 5|]
let left2, right2 = xs |> Array.splitAt 1   // [|1|] and [|2; 3; 4; 5|]
let left3, right3 = xs |> Array.splitAt 5   // [|1; 2; 3; 4; 5|] and [|||]
let left4, right4 = xs |> Array.splitAt 6   // InvalidOperationException
```

splitInto (Array, List, Seq)

Splits the input collection into at most count chunks.

```
// [[1; 2; 3; 4]; [5; 6; 7]; [8; 9; 10]]
// note that the first chunk has four elements
[1..10] |> List.splitInto 3

// [[1; 2; 3]; [4; 5; 6]; [7; 8]; [9; 10]]
[1..10] |> List.splitInto 4

// [[1; 2]; [3; 4]; [5; 6]; [7; 8]; [9]; [10]]
[1..10] |> List.splitInto 6
```

Aggregate functions

Aggregate functions take a collection of items and merge them into a smaller collection of items (often just one).

sum, average, min, max (Array, List, Seq)

All of these functions are specialized versions of a more generalized function *fold*.

```
let numbers = [1.0 .. 10.0]

let total = numbers |> List.sum          // 55.0
let average = numbers |> List.average    // 5.5
let max = numbers |> List.max            // 10.0
let min = numbers |> List.min            // 1.0
```

Miscellaneous functions

find (Array, List, Seq) == Single() in LINQ

find - finds the first element that matches a given condition.

```
let isDivisibleBy number elem = elem % number = 0

let input = [1 .. 10]

input |> List.find (isDivisibleBy 5)    // 5
input |> List.find (isDivisibleBy 11)   // KeyNotFoundException
```

findBack (Array, List, Seq)

```
let isDivisibleBy number elem = elem % number = 0

let input = [1 .. 10]

input |> List.findBack (isDivisibleBy 4)    // 8
input |> List.findBack (isDivisibleBy 11)   // KeyNotFoundException
```

findIndex (Array, List, Seq)

```
let isDivisibleBy number elem = elem % number = 0

let input = [1 .. 10]

input |> List.findIndex (isDivisibleBy 5)   // 4
input |> List.findIndex (isDivisibleBy 11)  // KeyNotFoundException
```

findIndexBack (Array, List, Seq)

```
let isDivisibleBy number elem = elem % number = 0

let input = [1..10]
```

```
input |> List.findIndexBack (isDivisibleBy 3)   // 8
input |> List.findIndexBack (isDivisibleBy 11)  // KeyNotFoundException
```

head, last, tail, item (Array, List, Seq)

Returns the first, last and all-but-first items in the collection.

```
let input = [15..22]

input |> List.head    // 15
input |> List.last    // 22
input |> List.tail    // [16; 17; 18; 19; 20; 21; 22]
```

item (Array, List, Seq)

Gets the element at a given index.

```
let input = [1..7]

input |> List.item 5      // 6
input |> List.item 8      // ArgumentException
```

take (Array, List, Seq)

Returns the elements of the collection up to a specified count.

```
let input = [1..10]

input |> List.take 5        // [1; 2; 3; 4; 5]
input |> List.take 11       // InvalidOperationException
```

truncate (Array, List, Seq)

Returns a collection that when enumerated returns at most N elements.

```
let input = [1..10]

input |> List.truncate 5    // [1; 2; 3; 4; 5]
input |> List.truncate 11   // [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

takeWhile (Array, List, Seq)

takeWhile returns each item in a new collection until it reaches an item that does not meet the predicate.

```
let input = [1..10]

input |> List.takeWhile (fun x -> x / 7 = 0)   // [1; 2; 3; 4; 5; 6]
input |> List.takeWhile (fun x -> x / 17 = 0)     // [1; 2; 3; 4; 5; 6; 7; 8; 9;
10]
```

skip (Array, List, Seq)

Returns a collection that skips N elements of the underlying sequence and then yields the remaining elements.

```
let input = [ for i in 1 .. 10 -> i * i ]  // [1; 4; 9; 16; 25; 36; 49; 64; 81;
100]

input |> List.skip 5                       // [36; 49; 64; 81; 100]
input |> List.skip 11                      // ArgumentException
```

skipWhile (Array, List, Seq)

Returns a collection, when iterated, skips elements of the underlying array (list, seq) while the given predicate returns true, and then yields the remaining elements.

```
let mySeq = seq { for i in 1 .. 10 -> i * i }    // 1 4 9 16 25 36 49 64 81 100

mySeq |> Seq.skipWhile (fun elem -> elem < 10)   // 16 25 36 49 64 81 100
mySeq |> Seq.skipWhile (fun elem -> elem < 101)  // Empty seq
```

exists (Array, List, Seq)

Tests if any element of the collection satisfies the given predicate.

```
let inputs = [0..3]

inputs |> List.exists (fun elem -> elem = 3)      // true
inputs |> List.exists (fun elem -> elem = 10)     // false
```

exists2 (Array, List, Seq)

Tests if any pair of corresponding elements of the collections satisfies the given predicate. The collections must have equal lengths, except for Seq where extra elements are ignored.

```
let list1to5 = [1 .. 5]           // [1; 2; 3; 4; 5]
let list0to4 = [0 .. 4]           // [0; 1; 2; 3; 4]
let list5to1 = [5 .. -1 .. 1]     // [5; 4; 3; 2; 1]
let list6to1 = [6 .. -1 .. 1]     // [6; 5; 4; 3; 2; 1]

(list1to5, list5to1) ||> List.exists2 (fun i1 i2 -> i1 = i2)    // true
(list1to5, list0to4) ||> List.exists2 (fun i1 i2 -> i1 = i2)    // false
(list1to5, list6to1) ||> List.exists2 (fun i1 i2 -> i1 = i2)    //
ArgumentException
```

forall (Array, List, Seq)

Tests if all elements of the collection satisfy the given predicate.

```
let inputs = [2; 4; 6; 8; 10]

inputs |> List.forall (fun i -> i % 2 = 0)    // true
inputs |> List.forall (fun i -> i % 2 = 0)    // false
```

forall2 (Array, List, Seq)

Returns true if all corresponding elements of the collection satisfy the given predicate pairwise. The collections must have equal lengths, except for Seq where extra elements are ignored.

```
let lst1 = [0; 1; 2]
let lst2 = [0; 1; 2]
let lst3 = [2; 1; 0]
let lst4 = [0; 1; 2; 3]

(lst1, lst2) ||> List.forall2 (fun i1 i2 -> i1 = i2)    // true
(lst1, lst3) ||> List.forall2 (fun i1 i2 -> i1 = i2)    // false
(lst1, lst4) ||> List.forall2 (fun i1 i2 -> i1 = i2)    // ArgumentException
```

contains (Array, List, Seq)

Returns true if a collection contains an equal value:

```
let rushSet = ["Dirk"; "Lerxst"; "Pratt"]
let gotSet = rushSet |> List.contains "Lerxst"      // true
```

filter, where (Array, List, Seq)

Returns a new collection containing only the elements of the collection for which the given predicate returns true.

```
let data =
    [("Cats",4)
     ("Tiger",5)
     ("Mice",3)
     ("Elephants",2) ]

// [("Cats", 4); ("Mice", 3)]
data |> List.filter (fun (nm, x) -> nm.Length <= 4)
data |> List.where (fun (nm, x) -> nm.Length <= 4)
```

length (Array, List, Seq)

Returns the length of the collection.

```
[ 1 .. 100 ] |> List.length         // 100
[ ] |> List.length                  // 0
[ 1 .. 2 .. 100 ] |> List.length    // 50
```

distinctBy (Array, List, Seq)

Returns a collection that contains no duplicate entries according to the generic hash and equality comparisons on the keys returned by the given key-generating function.

```
let inputs = [-5 .. 10]

// [-5; -4; -3; -2; -1; 0; 6; 7; 8; 9; 10]
inputs  |> List.distinctBy (fun i -> abs i)
```

distinct (Array, List, Seq) == Distinct() in LINQ

Returns a collection that contains no duplicate entries according to generic hash and equality comparisons on the entries.

```
[1; 3; 9; 4; 3; 1] |> List.distinct    // [1; 3; 9; 4]
[1; 1; 1; 1; 1; 1] |> List.distinct     // [1]
[ ] |> List.distinct                    // error FS0030: Value restriction
```

sortBy (Array, List, Seq) == OrderBy() in LINQ

Sorts the given collection using keys given by the given projection. Keys are compared using *Operators.compare*.

```
[1; 4; 8; -2; 5] |> List.sortBy (fun x -> abs x)    // [1; -2; 4; 5; 8]
```

sort (Array, List, Seq)

Sorts the given list using *Operators.compare*.

```
[1; 4; 8; -2; 5] |> List.sort    // [-2; 1; 4; 5; 8]
```

sortByDescending (Array, List, Seq)

```
[-3..3] |> List.sortByDescending (fun x -> abs x)   // [-3; 3; -2; 2; -1; 1; 0]
```

sortDescending (Array, List, Seq)

```
[0..5] |> List.sortDescending    // [5; 4; 3; 2; 1; 0]
```

sortWith (Array, List, Seq)

Sorts the given collection using the given comparison function.

```
let lst = ["<>"; "&"; "&&"; "&&&"; "<"; ">"; "|"; "||"; "|||"]

let sortFunction (str1: string) (str2: string) =
    if (str1.Length > str2.Length) then
        1
    else
        -1

// ["|"; ">"; "<"; "&"; "||"; "&&"; "<>"; "|||"; "&&&"]
lst |> List.sortWith sortFunction
```

Array, List and Seq functions

allPairs (Array, List, Seq)

Takes 2 arrays (list, seq), and returns all possible pairs of elements.

```
let arr1 = [| 0; 1 |]
let arr2 = [| 4; 5 |]

(arr1, arr2) ||> Array.allPairs arr1 arr2    // [|(0, 4); (0, 5); (1, 4); (1, 5)|]
```

append (Array, List, Seq)

Combines 2 arrays (list, seq).

```
let list1 = [33; 5; 16]
let list2 = [42; 23; 18]

List.append list1 list2     // [33; 5; 16; 42; 23; 18]
```

averageBy (Array, List, Seq)

averageBy take a function as a parameter, and this function's results are used to calculate the values for the average.

```
// val avg1 : float = 2.0
[1..3] |> List.averageBy (fun elem -> float elem)

// val avg2 : float = 4.666666667
[| 1..3 |] |> Array.averageBy (fun elem -> float (elem * elem))
```

Seq.cache

Seq.cache creates a stored version of a sequence. Use *Seq.cache* to avoid reevaluation of a sequence, or when you have multiple threads that use a sequence, but you must make sure that each element is acted upon only one time. When you have a sequence that is being used by multiple threads, you can have one thread that enumerates and computes the values for the original sequence, and remaining threads can use the cached sequence.

choose (Array, List, Seq)

choose enables you to transform and select elements at the same time.

```
let list1 = [33; 5; 16]

// [34; 6; 17]
list1 |> List.choose (fun elem -> Some(elem + 1))
```

compareWith (Array, List, Seq)

Compare two arrays (lists, seq) by using the *compareWith* function. The function compares successive elements in turn, and stops when it encounters the first unequal pair. Any additional elements do not contribute to the comparison.

```

```

```
let sq1 = seq { 1; 2; 4; 5; 7 }
let sq2 = seq { 1; 2; 3; 5; 8 }
let sq3 = seq { 1; 3; 3; 5; 2 }

let compareSeq seq1 seq2 =
    (seq1, seq2 ||> Seq.compareWith (fun e1 e2 ->
        if e1 > e2 then 1
        elif e1 < e2 then -1
        else 0)

let compareResult1 = compareSeq sq1 sq2     // int = 1
let compareResult2 = compareSeq sq2 sq3     // int = -1
```

concat (Array, List, Seq)

concat is used to join any number of arrays (lists, seq).

```
// int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
List.concat [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]]
```

Array.copy

Creates a new array that contains elements that are copied from an existing array. **The copy is a shallow copy**, which means that if the element type is a reference type, only the reference is copied, not the underlying object.

```
let firstArray : StringBuilder array = Array.init 3 (fun index -> new
StringBuilder(""))
let secondArray = Array.copy firstArray

firstArray.[0] <- new StringBuilder("Test1")
firstArray[0] <- new StringBuilder("Test1") // F# 6

firstArray.[1].Insert(0, "Test2") |> ignore
firstArray[1].Insert(0, "Test2") |> ignore // F# 6
```

Acknowledgments

Thanks goes to these people/projects:

- dungpa/fsharp-cheatsheet
- artag/fsharp-cheatsheet
- thriuin/fsharp-cheatsheet

F# vs C# Cheat Sheet

- Original source
- Referenced from FSharp Cheatsheet

Classes with properties and default constructor

```
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y

// Usage:
let v = Vector(10., 10.)
let x = v.X
let y = v.Y
```

```
public class Vector
{
    private double x;
    private double y;

    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public double X
    {
        get =>  { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
}

// Usage:
var v = new Vector(10, 10);
var x = v.X;
var y = v.Y;
```

Structs with properties

```
[<Struct>]
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y

// Usage:
let v = Vector(10., 10.)
let x = v.X
let y = v.Y
```

```
public struct Vector
{
    double x;
    double y;

    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double X
    {
        get { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
}

// Usage:
Vector v = new Vector(10, 10);
double x = v.X;
double y = v.Y;
```

Multiple constructors

```
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y
    new(v : Vector, s) = Vector(v.X * s, v.Y * s)

// Usage:
let v = Vector(10., 10.)
let w = Vector(v, 0.5)
```

```
public class Vector
{
    double x;
    double y;
    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public Vector(Vector v, double s) :
        this(v.x * s, v.y * s)
    {
    }
}

// Usage:
Vector v = new Vector(10, 10);
Vector w = new Vector(v, 0.5);
```

Member functions

```
type Vector(x : float, y : float) =
    member this.Scale(s : float) =
        Vector(x * s, y * s)

// Usage:
let v = Vector(10., 10.)
let v2 = v.Scale(0.5)
```

```
public class Vector
{
    double x;
    double y;

    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double Scale(double s)
    {
        return new Vector(this.x * s,
            this.y * s);
    }
}

// Usage:
```

```
Vector v = new Vector(10, 10);
Vector v2 = v.Scale(0.5);
```

Operators

```
type Vector(x, y) =
    member this.X = x
    member this.Y = y
    static member (*) (a : Vector, b : Vector) =
        a.X * b.X + a.Y + b.Y

// Usage:
let x = Vector(2., 2.)
let y = Vector(3., 3.)
let dp = x * y
```

```
public class Vector
{
    double x;
    double y;
    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double X
    {
        get { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
    public static double operator * (
        Vector v1, Vector v2)
    {
        return v1.x * v2.x + v1.y * v2.y;
    }
}

// Usage:
Vector x = new Vector(2, 2);
Vector y = new Vector(3, 3);
double dp = x * y;
```

Static members and properties

```
type Vector(x, y) =
    member this.X = x
    member this.Y = y
    static member Dot(a : Vector, b : Vector) =
        a.X * b.X + a.Y + b.Y
    static member NormX = Vector(1., 0.)

// Usage:
let x = Vector(2., 2.)
let y = Vector.NormX
let dp = Vector.Dot(x, y)
```

```
public class Vector
{
    double x;
    double y;
    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double X
    {
        get { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
    public static double Dot(Vector v1,
        Vector v2)
    {
        return v1.x * v2.x + v1.y * v2.y;
    }
    public static Vector NormX
    {
        get { return new Vector(1, 0); }
    }
}

// Usage:
Vector x = new Vector(2, 2);
Vector y = Vector.NormX;
double dp = Vector.Dot(x, y);
```

Class properties that use let value computations in the constructor

```
type Vector(x, y) =
    let mag = sqrt(x * x + y * y)
    let rad = if x = 0. && y = 0. then
                  0.
              else if x >= 0. then
                  asin(y / mag)
              else
                  (-1. * asin(y / mag)) +
                      Math.PI
    member this.Mag = mag
    member this.Rad = rad

// Usage:
let v = Vector(10., 10.)
let mag = v.Mag
let rad = v.Rad
```

```
public class Vector
{
    double mag = 0.0;
    double rad = 0.0;
    public Vector(double x, double y)
    {
        this.mag = Math.Sqrt(x * x + y * y);
        if (x == 0.0 && y == 0.0) rad = 0.0;
        else if (x >= 0.0)
            rad = Math.Asin(y / mag);
        else
            rad = (-1.0 * Math.Asin(y / mag)) + Math.PI;
    }
    public double Mag
    {
        get { return this.mag; }
    }
    public double Rad
    {
        get { return this.rad; }
    }
}

// Usage:
Vector v = new Vector(10, 10);
double mag = v.Mag;
double rad = v.Rad;
```

Class members that use private function values

```
type Vector(x, y) =
    let rotate a =
        let x' = x * cos a - y * sin a
        let y' = y * sin a + y * cos a
        new Vector(x', y')
    member this.RotateByDegrees(d) =
        rotate (d * Math.PI / 180.)
    member this.RotateByRadians(r) =
        rotate r

// Usage:
let v = Vector(10., 0.)
let x = v.RotateByDegrees(90.)
let y = v.RotateByRadians(Math.PI / 6.)
```

```
public class Vector
{
    double x = 0.0;
    double y = 0.0;
    Vector rotate(double a)
    {
        double xx = this.x * Math.Cos(a) -
                    this.y * Math.Sin(a);
        double yy = this.y * Math.Sin(a) +
                    this.y * Math.Cos(a);
        return new Vector(xx, yy);
    }
    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public Vector RotateByDegrees(double d)
    {
        return rotate(d * Math.PI / 180);
    }
    public Vector RotateByRadians(double r)
    {
        return rotate(r);
    }
}

// Usage:
Vector v = new Vector(10, 10);
Vector x = v.RotateByDegrees(90.0);
Vector y = v.RotateByRadians(Math.PI / 6.0);
```

Overloading members

```
type Car() =
    member this.Drive() =
        this.Drive(10)
        ()
    member this.Drive(mph : int) =
        // Do something
        ()

// Usage:
let c = Car()
c.Drive()
c.Drive(10)
```

```
public class Car
{
    public void Drive()
    {
        Drive(10);
    }
    public void Drive(int mph)
    {
        // Do something
    }
}

// Usage:
Car c = new Car();
c.Drive();
c.Drive(10);
```

Mutable fields in a class with get/set properties

```
type MutableVector(x : float, y : float) =
    let mutable cx = x
    let mutable cy = y
    member this.X with get() = cx and
                       set(v) = cx <- v
    member this.Y with get() = cy and
                       set(v) = cy <- v
    member this.Length = sqrt(x * x + y * y)

// Usage:
let v = MutableVector(2., 2.)
```

```
let len1 = v.Length
v.X <- 3.
v.Y <- 3.
let len2 = v.Length
```

```
public class MutableVector
{
    public MutableVector(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }
    public double X { get; set; }
    public double Y { get; set; }
    public double Length
    {
        get { return Math.Sqrt(
                  this.X * this.X +
                  this.Y * this.Y); }
    }
}

// Usage:
MutableVector v = new MutableVector(2.0, 2.0);
double len1 = v.Length;
v.X = 3.0;
v.Y = 3.0;
double len2 = v.Length;
```

Generic classes and function arguments

```
type Factory<'T>(f : unit -> 'T) =
    member this.Create() =
        f()

// Usage:
let strings = Factory<string>(
    fun () -> "Hello!")
let res = strings.Create()
let ints = Factory<int>(fun () -> 10)
let res = ints.Create()
```

```
public class Factory<T>
{
    Func<T> creator;
```

```
    public Factory(Func<T> f)
    {
        this.creator = f;
    }
    public T Create()
    {
        return this.creator();
    }
}

// Usage:
var strings = new
    Factory<string>(() => "Hello");
var res1 = strings.Create();
var ints = new Factory<int>(() => 10);
var res2 = ints.Create();
```

Generic classes and methods

```
type Container<'T>(a : 'T) =
    member this.Convert<'U>(f : 'T -> 'U) =
        f a

// Usage:
let c = new Container<int>(10)
let b = c.Convert(fun a -> a.ToString())
```

```
public class Container<T>
{
    private T value;
    public Container(T t)
    {
        this.value = t;
    }
    public U Convert<U>(Func<T, U> f)
    {
        return f(this.value);
    }
}

// Usage:
Container<int> c = new Container<int>(10);
string result = c.Convert(n => n.ToString());
```

Extension methods

```
type List<'T> with
    member this.MyExtensionMethod() =
        this |> Seq.map (fun a -> a.ToString())

// Usage:
let c = [1; 2; 3]
let d = c.MyExtensionMethod()
```

```
public static class ExtensionMethods
{
    public static IEnumerable<string>
        MyExtensionMethod<T>(this List<T> a)
    {
        return a.Select(s => s.ToString());
    }
}

// Usage:
List<int> l = new List<int> { 1, 2, 3 };
IEnumerable<string> res =
    l.MyExtensionMethod();
```

Extension properties

```
type List<'T> with
    member this.MyExtensionProp =
        this |> Seq.map (fun a -> a.ToString())

// Usage:
let c = [1; 2; 3]
let d = c.MyExtensionProp
```

```
// N/A. C# does not support this feature.
```

Indexers

```
type Table() =
    member this.Item
        with get(key : string) = int key
```

```
// Usage:
let tab = Table()
let x = tab.["10"]
let y = tab.["12"]
```

```
public class Table
{
    public int this[string key]
    {
        get { return Convert.ToInt32(key); }
    }
}

// Usage:
Table tab = new Table();
int x = tab["10"];
int y = tab["12"];
```

Indexed Properties

```
type Table() =
    member this.Values
        with get(key : string) = int key
    member this.MultipleValues
        with get(key1, key2) = key1 + key2
// Usage:
let tab = Table()
let x = tab.Values("10")
let y = tab.Values("12")
let a = tab.MultipleValues(10, 5)
```

```
// N/A. C# does not support this feature.
```

Abstract classes

```
[<AbstractClass>]
type Shape() =
    abstract Name : string with get
    abstract Scale : float -> Shape
```

```
public abstract class Shape
{
    public abstract string Name { get; }
    public abstract Shape Scale(double scale);
}
```

Derive from a base class and overriding base methods with generics

```
[<AbstractClass>]
type Shape<'T>() =
    abstract Name : string with get
    abstract Scale : float -> 'T

type Vector(angle, mag) =
    inherit Shape<Vector>()
    member this.Angle = angle
    member this.Mag = makg
    override this.Name = "Vector"
    override this.Scale(factor) =
        Vector(angle, mag * factor)

// Usage:
let v = Vector(45., 10.)
let v2 = v.Scale(0.5)
```

```
public abstract class Shape<T>
{
    public abstract string Name { get; }
    public abstract T Scale(double scale);
}

public class Vector : Shape<Vector>
{
    double angle;
    double mag;
    public double Angle {get{return angle;}}
    public double Mag {get{return mag;}}
    public Vector(double angle, double mag)
    {
        this.angle = angle;
        this.mag = mag;
    }
    public override string Name
    {
        get { return "Vector"; }
    }
    public override Vector Scale(double scale)
```

```
    {
        return new Vector(this.Angle,
            this.Mag * scale);
    }
}
// Usage:
Vector v = new Vector(45, 10);
Vector v2 = v.Scale(0.5);
```

Calling a base class method

```
type Animal() =
    member this.Rest() =
        // Rest for the animal
        ()

type Dog() =
    inherit Animal()
    member this.Run() =
        // Run
        base.Rest()
// Usage:
let brian = new Dog()
brian.Run()
```

```
public class Animal
{
    public void Rest()
    {
        // Rest for the animal
    }
}
public class Dog : Animal
{
    public void Run()
    {
        // Run
        base.Rest();
    }
}
// Usage:
Dog brian = new Dog();
brian.Run();
```

Implementing an interface

```
type IVector =
    abstract Mag : double with get
    abstract Scale : float -> IVector

type Vector(x, y) =
    interface IVector with
        member this.Mag = sqrt(x * x + y * y)
        member this.Scale(s) =
            Vector(x * s, y * s) :> IVector
    member this.X = x
    member this.Y = y
// Usage:
let v = new Vector(1., 2.) :> IVector
let w = v.Scale(0.5)
let mag = w.Mag
```

```
interface IVector
{
    double Mag { get; }
    IVector Scale(double s);
}
class Vector : IVector
{
    double x;
    double y;
    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double X
    {
        get { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
    public double Mag
    {
        get { return Math.Sqrt( this.x * this.x +
                this.y * this.y); }
    }
    public IVector Scale(double s)
    {
        return new Vector(this.x * s,
            this.y * s);
    }
}
// Usage:
```

```
IVector v = new Vector(1, 2);
IVector w = v.Scale(0.5);
double mag = w.Mag;
```

Implementing an interface with object expressions

```
type ICustomer =
    abstract Name : string with get
    abstract Age : int with get

let CreateCustomer name age =
    { new ICustomer with
        member this.Name = name
        member this.Age = age
    }
// Usage:
let c = CreateCustomer "Snoopy" 16
let d = CreateCustomer "Garfield" 5
```

```
/// N/A. C# does not support creating object expressions.
```

Events

```
type BarkArgs(msg:string) =
    inherit EventArgs()
    member this.Message = msg

type BarkDelegate =
    delegate of obj * BarkArgs -> unit

type Dog() =
    let ev = new Event<BarkDelegate, BarkArgs>()
    member this.Bark(msg) =
        ev.Trigger(this, new BarkArgs(msg))
    [<CLIEvent>]
    member this.OnBark = ev.Publish
// Usage:
let snoopy = new Dog()
snoopy.OnBark.Add(
    fun ba -> printfn "%s" (ba.Message))
snoopy.Bark("Hello")
```

```
public class BarkArgs : EventArgs
{
    private string msg;
    public BarkArgs(string msg)
    {
        this.msg = msg;
    }
    public string Message
    {
        get { return this.msg; }
    }
}

public delegate void BarkDelegate(
    Object sender, BarkArgs args);

class Dog
{
    public event BarkDelegate OnBark;
    public void Bark(string msg)
    {
        OnBark(this, new BarkArgs(msg));
    }
}
// Usage:
Dog snoopy = new Dog();
snoopy.OnBark += new BarkDelegate(
    (sender, msg) =>
        Console.WriteLine(msg.Message));
snoopy.Bark("Hello");
```

Explicit class constructor

```
type Vector =
    val mutable x : float
    val mutable y : float
    new() = {x = 0.; y = 0.}

// Usage:
let v = Vector()
v.x <- 10.
v.y <- 10.
```

Explicit public fields

```
type Vector() =
    [<DefaultValue()>]
    val mutable x : int
    [<DefaultValue()>]
    val mutable y : int

// Usage:
let v = Vector()
v.x <- 10
v.y <- 10
```

```
public class Vector
{
    public int x;
    public int y;
}

// Usage:
Vector v = new Vector();
v.x = 10;
v.y = 10;
```

Explicit struct definition

```
[<Struct>]
type Vector =
    val mutable public x : int
    val mutable public y : int

// Usage:
let mutable v = Vector()
v.x <- 10
v.y <- 10
```

```
public struct Vector
{
    public int x;
    public int y;
}

// Usage:
Vector v = new Vector();
v.x = 10;
v.y = 10;
```